Massively Parallel Algorithms for Real-Time Wavefront Control of a Dense Adaptive Optics System

Amir Fijany, Mark Milman, and David Redding

Jet propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91109

ABSTRACT

In this paper massively parallel algorithms and architectures for real-time wavefront control of a dense adaptive optic system (SELENE) are presented. We have already shown that the computation of a near optimal control algorithm for SELENE can be reduced to the solution of a discrete Poisson equation on a regular domain. Although, this represents an optimal computation, due the large size of the system and the high sampling rate requirement, the implementation of this control algorithm poses a computationally challenging problem since it demands a sustained computational throughput of the order of 10 GFlops. We develop a novel algorithm, designated as Fast Invariant Imbedding algorithm, which offers a massive degree of parallelism with simple communication and synchronization requirements. Due to these features, our algorithm is significantly more efficient than other Fast Poisson Solvers for implementation on massively parallel architectures. We also discuss two massively parallel, algorithmically specialized, architectures for low-cost and optimal implementation of the Fast Invariant Imbedding algorithm.

1. INTRODUCTION

This paper presents massively parallel algorithms and architectures for wavefront control of the Space Laser Electric Energy (SELENE) power beaming system. The real-time wavefront control of SELENE represents a computationally challenging problem due to the large size of the system and the high sampling rate requirement. We have already shown that the computation of a novel, near optimal, control wavefront algorithm can be reduced to the solution of a two-dimensional (2D) discrete Poisson equation with Neumann boundary conditions [11. Although, SELENE can have rather arbitrary geometries, we have shown [11 that the computational domain of the corresponding discrete Poisson equation can be transformed into a regular square domain.

Such a domain regularization strategy, while introduces a minimal error in the computation, enable the use of the so called Fast Poisson Solvers [2] with an optimal computational cost for our problem. For a typical configuration of the SELENE comprising a 2D array of NxN segments, the computational complexity of our control strategy is then of $O(N^2 \text{Log } N)$. Compared with the $O(N^4)$ computational complexity of fully optimal control strategies and for the typical values of N of the order of hundreds, this represents more than three orders of magnitude improvement in the computational cost. However, even with such a drastic improvement, the real-time implementation of our control

strategy by using the Fast Poisson Solvers still requires a formidable computing capability. For a typical case of N = 500 and with a 1 KHz sampling rate, the real-time implementation will require a sustained computational throughput of the order of 10 GFlops. This clearly suggests that the exploitation of a massive degree of parallelism is the key factor for achieving the required computational efficiency in a cost-effective fashion.

Swarztrauber and Sweet [3] have presented an extensive comparative analysis of efficiency of various Fast Poisson Solvers for implementation on vector and parallel architectures. This study suggests that the Matrix-Decomposition (01, Fourier Analysis [1,4]) algorithm is the most efficient for a realistic parallel implementation by using a number of processors of the order of hundreds. However, the practical implementation of the MD algorithm (though for a three-dimensional problem [51) has shown that the resulting communication cost can significantly reduce its efficiency for parallel computation. A more extensive analysis of communication complexity of the MD algorithm is presented in §4.

The Invariant Imbedding algorithm [6,7] was one of the earliest methods for direct. solution of the Poisson equation. However, with the development the Fast Poisson Solvers with a much greater efficiency, it seems that less attention is paid to this algorithm. In this paper we develop a novel variant of this algorithm, designated by Fast Invariant Imbedding algorithm, which achieves the same computational efficiency as the best Fast Poisson Solver. The main advantage of the Fast Invariant Imbedding algorithm over the other Fast Poisson Solvers and particularly the MD algorithm is that it is significantly more efficient for parallel computation, In fact, the simple communication and synchronization requirements of our algorithm enables its efficient implementation on a variety of parallel and vector architectures.

This paper is organized as follows, In §2, the Dirichlet problem for the Poisson equation and the MD algorithm are reviewed. In §3, we first review the the original Invariant Imbedding algorithm. We then develop the Fast Invariant Imbedding and discuss its extension to solution of the problem with Neumann boundary as well as its excellent numerical properties, In §4, the parallel implementation of the Fast Invariant Imbedding algorithm on various parallel architectures is discussed and its performance is compared with that of the MD algorithm. We also present a hybrid parallel/pipeline strategy for efficient computation of our algorithm along with two algorithmically specialized parallel architectures for its optimal and cost-effective implementation.

2. TWO-DIMENSIONAL POISSON EQUATION AND MATRIX-DECOMPOSITION ALGORITHM

2.1. Dirichlet Problem

We consider the Dirichlet problem for the two-dimensional (2D) Poisson equation in a unit square domain Ω with boundary $\partial\Omega$ as

$$\nabla^2 u(x,y) = f(x,y) \qquad (x,y)\in\Omega \tag{2.1}$$

$$u(x,y) = g(x,y) \qquad (x,y)\in\partial\Omega$$

Superimposing a uniform mesh of size $\Delta x = \Delta y = 1/(N+1)$ and using the five-point finite-difference approximation, the problem is reduced to solution of a linear system

$$Mu = W \tag{2.2}$$

for U where

$M \in \mathbb{R}^{N^2 \times N^2}$ is a block tridiagonal matrix given by M = Tridiag[-I, B, -I], I is the NxN identity matrix,

$B \in \mathbb{R}^{N \times N}$ is a tridiagonal matrix given by B = Tridiag[-1, 4, -1],

$u = Col\{U_i\} \in \mathbb{R}^{N^2}$, i = 1 to N, and $U_i = Col\{U_{ij}\} \in \mathbb{R}^N$, j = 1 to N, is the vector representing the approximate solution for u(x,y), and

$w = Col\{W_i\} \in \mathbb{R}^{N^2}$, i = 1 to N, and $W_i = Col\{w_{ij}\} \in \mathbb{R}^N$, j = 1 to N, is the vector resulting from the discretization of f(x,y) and g(x,y).

Alternatively, we present vectors of dimension N^2 by NxN matrices. To this end, the matrix representation of U and W are denoted by $\underline{U}$ and $\underline{W}$ where $\underline{U} \triangleq \{U_{ij}\}$ and $\underline{W} \triangleq \{W_{ij}\} \in \mathbb{R}^{N \times N}$, i and j = 1 to N.

2.2. Matrix Decomposition Algorithm

The MD algorithm is based on a specific decomposition of matrix M. Following theorem is used in the derivation of the algorithm.

Theorem 1. The Eigenvalue-Eigenvector (E-E) decomposition of a symmetric tridiagonal toeplitz matrix S = Tridiag[b, a, b] $\in \mathbb{R}^{N \times N}$ is given as

$$S = Q \lambda_S Q$$

The matrix $Q = \{Q_{ij}\} \in \mathbb{R}^{N \times N}$, i and j = 1 to N, is the set of normalized eigenvectors of S with $Q_{ij} = (2/N+1)^{1/2}(\sin(ij\pi/N+1))$. The diagonal matrix $\lambda_S = Diag\{\lambda_{Si}\} \in \mathbb{R}^{N \times N}$ is the set of eigenvalues with $\lambda_{Si} = a + 2b\cos(i\pi/N+1)$ being the ith eigenvalue.

Proof. See for example Barnett [8].

Note that, Q is a symmetric orthonormal matrix and hence $Q = Q^{-*} = Q^t$ (t denotes the transpose).

Let us define a matrix $\mathcal{Q} \triangleq Diag[Q, Q, \ldots, Q, Q] \in \mathbb{R}^{N^2 \times N^2}$. From its definition, it follows that $\mathcal{Q}$ is a symmetric orthonormal matrix and hence $\mathcal{Q} = \mathcal{Q}^t = \mathcal{Q}^{-1}$.

Also, consider a symmetric permutation matrix $P \in \mathbb{R}^{N^2 \times N^2}$ that arises in 2-D Discrete Fourier Transform (DFT). If two vectors V and R of dimension N^2 are defined as $V = Col\{V_i\}$ and $R = Col\{R_i\}$, i = 1 to N, and $V_i = Col\{V_{ij}\}$ and $R_i = Col\{R_{ij}\}$, j = 1 to N, then V = PR implies that $V_{i,j} = R_{j,i}$. Or, using matrix representation of V and R, we have

$$V = PR \Rightarrow \underline{V} = \underline{R}^t$$

That is, P is the operator for matrix transposition. We also have $P^{-1} = P^t$,

since P 3s a permutation matrix and hence it is orthogonal, and $P = P^t = P^{-1}$, since P is symmetric.

Theorem 2. The matrix M has a decomposition as

$$M = QPTPQ \tag{2.3}$$

where T is a block diagonal matrix given below.

Proof. From Theorem 1, the E-E decomposition of B is given as $B = Q\lambda_B Q$ where $\lambda_B = Diag\{\lambda_{Bi}\} \epsilon R^{NxN}$, i = 1 to N, and $\lambda_{Bi} = 4 - 2\cos(i\pi/N+1)$. Using the E-E decomposition of B, the matrix M can be expressed by

$$M = Tridiag[-I, Q\lambda_B Q, -I] = QAQ \tag{2.4}$$

where A is a block tridiagonal matrix as $A = Tridiag[-I, \lambda_{B'} -11$, The block elements of A are diagonal and hence the matrix A can be reduce to a block diagonal matrix as

$$A = PPAPP = P(PAP)P = PTP \tag{2.5}$$

where $T = Diag\{T_i\}$ and $T_i = Tridiag[-1, \lambda_{Bi}, -1]$. The decomposition of M, given by (2.3), follows by substituting [2.5) into (2.4). □

The MD algorithm is derived by substituting the decomposition of matrix M into (2.2). The computation of the MD algorithm is performed as follows.

Step 1: Compute $\tilde{W} = QW$ or $\tilde{W}_i = QW_i$ for i = 1 to N.

Step 2: Form vector $\hat{W} = P\tilde{W}$ or $\hat{\underline{W}} = \tilde{\underline{W}}^t$, i.e., $W_{i,} = \tilde{W}_{j,i}$ for i and j = 1 to N.

Step 3: Solve the tridiagonal systems $T_i\hat{U}_i = \hat{W}_i$ for i = 1 to N

Step 4: Form vector $\tilde{U} = P\hat{U}$ or $\tilde{\underline{U}} = \hat{\underline{U}}^t$, i.e., $\tilde{U}_{i,j} = \hat{U}_{j,i}$ for i and j = 1 to N

Step 5: Compute $U = Q\tilde{U}$ or $U_i = Q\tilde{U}_i$ for i = 1 to N.

The matrix Q is the operator of 1-D Discrete Sine Transform (DST). Thus, by using fast techniques [9], the matrix-vector multiplications in Steps 1 and 5 can be performed in $O(N\text{Log } N)$. This leads to a computational complexity of $0(N^2\text{Log } N)$ **for Steps 1 and 5.** The cost of each tridiagonal linear system solution in Step 3 is of O(N) which leads to a cost of $O(N^2)$ for this step,

3. THE INVARIANT IMBEDDING ALGORITHM

3.1. The Original Invariant Imbedding Algorithm

The Invariant Imbedding algorithm [6,7] is based on the observation that the solution of (2.2) is equivalent to that of a discrete two-point boundary-value problem given by

$$-U_{i-1} + BU_i - U_{i+1} = W_i \qquad i = 1 \text{ to } N \tag{3.1}$$

with boundary values U_0 and U_{N+1}. Note that, U_0 and U_{N+1} are given through the

specification of boundary conditions in (2. 1). A solution to (3.1) is then sought of the form:

$$u_{i+1} = A_i U_i + R_i \tag{3.2}$$

where matrices A_i's and vectors R_i's are independent of U,'s. From (3, 1) and (3.2), it follows that

$$u, = (B - A_i)^{-1} U_{i-1} + (B - A_i)^{-1}(R_i + W_i) \tag{3.3}$$

from which the recurrences for computation of A_i and R_i are derived as

$$A_{1-1} = [B - A_i)^{-1} \tag{3.4}$$

$$R_{i-1} = (B - A_i)^{-1}(R_i + W_i) = A_{i-1}(R_i + W_i) \tag{3.5}$$

The initial conditions for the above recurrences are obtained by considering (3.2) for i = N which implies that $A_N = O$ and $R_N = U_{N+1}$. As is shown in [6,7], from positive definiteness of B it follows that the matrices $(B - A_i)$ are also positive definite and hence nonsingular.

The computation of Invariant Imbedding algorithm is performed as follows.

Step 1: Compute $A_{i\,1}$ from (3.4) for i = N to 1 with $A_N = O$.

Step 2: Compute $R_{i\,1}$ from (3.5) for i = N to 1 with $R_N = U_{N+1}$.

Step 3: Compute U_{i+1} from (3.2) for i = O to N-1 with U. given.

The computational complexity of Step 1 is of $O(N^4)$ while that of Steps 2 and 3 is of $O(N^3)$. This leads to an overall computational complexity of $O(N^4)$ for the algorithm. However, the matrices A_i's are only function of problem's size (i.e., N), the type of finite-difference scheme employed, and the type of boundary condition. Thus, for some cases, such as real-time control of SELENE, these matrices can be precomputed. With this precomputation, the computational cost of the algorithm is reduced to $O(N^3)$ which indicates that the algorithm is still less efficient than other Fast Poisson Solvers.

3.2. **A Fast Invariant Imbedding Algorithm**

The inefficiency of the Invariant Imbedding algorithm results from the fact that it requires the inversion of dense matrices A,'s. However, as shown below, these matrices have fast E-E decomposition which allows the diagonalization of (3.2), (3.4), and (3.5). This diagonalization results in an algorithm that not only it is highly competitive for sequential implementation but also it 1s very efficient for parallel and vector computation. The diagonalization procedure is based on the fact that the matrices A_i's have a same set of eigenvectors but different sets of eigenvalues. This is established by the following theorem.

Theorem 2. The E-E decomposition of matrix A_i is given as

$$A_i = Q\lambda_{Ai}Q \tag{3.6}$$

where $\lambda_{Ai} = \text{Diag}\{\lambda_{Ai\,j}\} \in R^{N\times N}$, j = 1 to N, is given below.

Proof. The proof follows by induction. From (3.4), for = N we have

$$A_{N-1} = B^{-1} = (Q\lambda_B Q)^{-1} = Q\lambda^{-1}Q$$

which implies that

$$\lambda_{AN-1} = \lambda_B^{-1}$$

Now let $A_{i+1} = Q\lambda_{Ai+1}Q$. From **(3.4),** it follows that

$$A_i = (B - A_{i+1})^{-1} = (Q\lambda_B Q - Q\lambda_{Ai+1}Q)^{-1} = Q(\lambda_B - \lambda_{Ai+1})^{-1}Q$$

The set of eigenvalues of matrices A_i's are then given by

$$\lambda_{Ai} = (\lambda_B - \lambda_{Ai+1})^{-1}, \quad i = N\text{-}1 \text{ to } 0, \text{ with } \lambda_{AN} = 0 \tag{3.7}$$

Substituting the E-E decomposition of A_i's, given by (3.6), into (3.2) and (3.5), and defining

$$\tilde{U}_i = QU_i, \ \tilde{R}_i = QR_i, \text{ and } \tilde{W}_i = QW_i$$

the fast variant of Invariant Imbedding algorithm is then given by

$$\tilde{R}_{i-1} = \lambda_{Ai-1}(\tilde{R}_i + \tilde{W}_i), \ i = N \text{ to } 1, \text{ with } \tilde{R}_N = \tilde{U}_{N+1} \tag{3.8}$$

$$\tilde{U}_{i+1} = \lambda_{Ai}\tilde{U}_i + \tilde{R}_i, \ i = 0 \text{ to } N\text{-}1, \text{ with given } \tilde{U}_0 \tag{3.9}$$

where λ_{Ai}'s are computed from (3."/). The efficiency of the algorithm can be further increased by avoiding the explicit computation of $\tilde{U}_0$ and $\tilde{U}_{N+1}$, i.e., by avoiding explicit transformation of U_0 and U_{N+1}. To this end, we rewrite (3.8) for i = N as

$$\tilde{R}_{N-1} = \lambda_{AN-1}(\tilde{R}_N + \tilde{W}_N) = \lambda_{AN-1}(\tilde{U}_{N+1} + \tilde{W}_N) = \lambda_{AN-1}\tilde{W}'_N$$

where $\tilde{W}'_N = QW'_N$ and $W'_N = U_{N+1} + W_N$. Similarly, we rewrite (3.8) for i = 0 as

$$\tilde{U}_1 = \lambda_{A0}\tilde{U}_0 + \tilde{R}_0 = \lambda_{A0}\tilde{U}_0 + \lambda_{A0}(\tilde{R}_1 + \tilde{W}_1) = \lambda_{A0}(\tilde{R}_1 + \tilde{W}'_1)$$

where $\tilde{W}'_1 = QW'_1$ and $W'_1 = U_0 + W_1$.

The computation of Fast Invariant Imbedding algorithm is performed as follows,

Step 1: Compute $A_{Ai} = (\lambda_B - A_{Ai+1})^{-1}$, i = N-1 to 0, with $A_{AN} = 0$.

Step 2: Compute $W'_1 = U_0 + W_1$, $W'_N = U_{N+1} + W_N$, and set $W'_i = W_i$, i = 2 to N-1.

Compute $\tilde{W}' = QW'$ or $\tilde{W}'_i = QW'_i$ for i = 1 to N.

Step 3: Compute $\tilde{R}_{i\,i}$ for i = N-1 to 1 from

$$\tilde{R}_{i-1} = \lambda_{Ai-1}(\tilde{R}_i + \tilde{W}'_i) \text{ with } \tilde{R}_{N-1} = \lambda_{AN-1}\tilde{W}'_N$$

Step 4: Compute $\tilde{U}_{i+1}$ for i = 1 to N-1 from

$$\tilde{U}_{i+1} = \lambda_{Ai}\tilde{U}_i + \tilde{R}_i \text{ with } \tilde{U}_1 = \lambda_{A0}(\tilde{R}_1 + \tilde{W}'_1)$$

Step 5: Compute $U = Q\tilde{U}$ or $u_i = Q\tilde{U}_i$ for i = 1 to N.

Note that, similar to matrices A_i's, the diagonal matrices λ_{Ai}'s are only function of problem's size, the type of finite-difference scheme, and boundary conditions, and hence for some cases they can be precomputed, However, in the following, the possibility of this precomputation is not considered in evaluating the computational cost of the algorithm.

The computational complexity of Steps 1, 3, and 4 is of $O(N^2)$. Except for the computation of W'_1 and W'_N, the computation of Step 2 and 5 are exactly the same as the Steps 1 and 4 of the MD algorithm. It follows that the Fast Invariant Imbedding algorithm is, asymptotically, as fast as the MD algorithm with the same constant for N^2Log N-dependent term, A more detailed comparison can show that the algorithm is also competitive in terms of the actual number of operations. Let *f* denote the cost of one floating-point operation. The cost of Steps 1, 3, and 4 is given by *6N2f.* Boisvert [101 has compared various algorithms for solution of symmetric tridiagonal toeplitz systems. Using the best ***general*** algorithm in [10] the cost of solving N tridiagonal systems in Step 3 of the MD algorithm is also given by *6N2f.*

3.3. **Numerical Properties of Invariant Imbedding Algorithms**

Both the original and Fast Invariant Imbedding algorithms have excellent numerical properties. Angel [6] has shown that the recurrence in (3.4) is stable in the sense that an error introduced at any stage of the calculation does not cause larger errors in the preceding stages and, asymptotically, it will be reduced to zero. It then follows that the recurrence in (3.7) is also stable since it is obtained from (3.4) through an orthogonal transformation. The two vector recurrences in (3.8) and (3.9) can be written as two sets of N scalar First-Order Inhomogeneous Linear Recurrences (FOILRs) as

$$\tilde{R}_{i-1,j} = A_{Ai-1,j}(\tilde{R}_{i,j} + \tilde{W}'_{i,j}),\ i = N\text{-}1 \text{ to } 1 \text{ and } j = 1 \text{ to } N \qquad (3.10)$$

$$\tilde{U}_{i+1,j} = \lambda_{Ai,j}\tilde{U}_{i,j} + \tilde{R}_{i,j},\ i = 1 \text{ to } N\text{-}1 \text{ and } j = 1 \text{ to } N \qquad (3,11)$$

Similarly, (3.7) can be written as a set of N scalar first-order nonlinear recurrences **as**

$$\lambda_{Ai,j} = \frac{1}{\lambda_{Bj} - \lambda_{Ai+1,j}},\ i = N\text{-}1 \text{ to } 0 \text{ and } j = N \text{ to } 1, \quad \text{with } \lambda_{AN,j} = 0 \qquad (3.12)$$

Note that, (3.12) represents a Continued Fraction which can be transformed to a second-order linear recurrence [11]. Since $\lambda_{Bj} > 2$ for all j = 1 to N, it can be then easily shown by induction that $1 > \lambda_{Aij} > 0$. This implies that the two sets of recurrences in (3. 10)-(3, 11) are stable in the sense that an error introduced at any stage of the calculation does not cause larger errors in the preceding stages and, asymptotically, it will be reduced to zero.

3.4. Neumann **Problem**

At first glance, the extension of both Invariant Imbedding algorithm and its fast variant to the solution of problem with other boundary conditions might seem less straightforward than that for other FPSs. For the Invariant Imbedding Method appears to be well suited for Dirichlet boundary condition for which U_0 and U_{N+1} are explicitly given, Angel and Bellman [7] have extended the Invariant Imbedding algorithm for Neumann-Dirichlet boundary condition. Here, we extend the Fast Invariant Imbedding algorithm to the solution of problem with a more generalized Neumann-Neumann boundary condition given by

$$\frac{\partial u}{\partial x}(0,y) = \phi_0(y) \text{ and } \frac{\partial u}{\partial x}(1,y) = \phi_1(y) \tag{3.13}$$

$$\frac{\partial u}{\partial y}(x,0) = \varphi_0(x) \text{ and } \frac{\partial u}{\partial y}(x,1) = \varphi_1(x) \tag{3.14}$$

where $\phi_0(y)$, $\phi_1(y)$, $\varphi_0(x)$, and $\varphi_1(x)$ are given functions.

Extending the domain by introducing fictitious points $u(-1,j)$ and $u(N+2,j)$, for j = 0 to N+1, (3.1) is now written as

$$-U_{i-1} + BU_i - U_{i+1} = W'_i \qquad i = 0 \text{ to } N+1 \tag{3.15}$$

where, now, U_i and $W'_i \varepsilon R^{(N+2)}$, and $B \varepsilon R^{(N+2)\times(N+2)}$ is a tridiagonal matrix as

$$B = \begin{bmatrix} 4 & -2 & & & \\ -1 & 4 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 4 & -1 \\ & & & -2 & 4 \end{bmatrix} \tag{3.16}$$

The structure of matrix B results from the discretization of the boundary condition in (3.13) by using a central-difference scheme with a second order accuracy. Also, the vectors W'_i, for i = 1 to N, include the contribution of $\phi_0(y)$ and $\phi_1(y)$. The expressions of W'_0 and W'_{N+1} are given below.

By using the results of [9, p. 252], it can be shown that the E-E decomposition of matrix B is given by:

$$B = Q\lambda_B Q^{-1} \tag{3.17}$$

The matrix $Q = \{Q_{ij}\} \varepsilon R^{(N+2)\times(N+2)}$, i and j = 0 to N+1, is the set of eigenvectors of B with $Q_{ij} = \cos(ij\pi/N+1)\}$ and $\lambda_B = \text{Diag}\{\lambda_{Bi}\} \varepsilon R^{(N+2)\times(N+2)}$, i = 0 to N+1, is the set of eigenvalues of B with $\lambda_{Bi} = 4 - 2\cos(i\pi/N+1)$ being the ith eigenvalue. Note that, the matrix Q is not orthogonal, However, as shown in *[9]*, Q and Q-l can be expressed as:

$$Q = \theta\mathcal{Y} \text{ and } Q\text{-}l = (2/N-1)\mathcal{Y}^{-1}\theta$$

where 0 is the 1-D Discrete Cosine Transform (DCT) operator and Y is a diagonal scaling matrix given by $\mathcal{Y}$ = [2, 1, ... 1, 2]. Defining the

normalized DCT operator as $\Theta = (2/N+2)^{1/2}\theta$, (3.17) can be then written as

$$B = \Theta\mathcal{Y}\lambda_B\mathcal{Y}^{-1}\Theta$$

Defining

$$\tilde{U}_i = \Theta\mathcal{Y}U_i \Rightarrow U_i = \mathcal{Y}^{-1}\Theta\tilde{U}_i$$

the diagonalized version of (3.15) is written as:

$$-\tilde{U}_{i-1} + \lambda_{Bi}\tilde{U}_i - \tilde{U}_{i+1} = \tilde{W}'_i \qquad i = 0 \text{ to } N+1 \tag{3.18}$$

Discretising (3.14) by using a central-difference scheme gives

$$U_{N+2} - U_N = \varphi_1 \Rightarrow \tilde{U}_{N+2} = \tilde{U}_N + \tilde{\varphi}_1 \tag{3.19}$$

$$U_1 - U_{-1} = \varphi_0 \Rightarrow \tilde{U}_{-1} = \tilde{U}_1 + \tilde{\varphi}_0 \tag{3.20}$$

where $\varphi_1 = \text{col}\{\varphi_1(x_j)\}$ and $\varphi_0 = \text{col}\{\varphi_0(x_j)\}$, j = 0 to N+1. From (3.18)-(3.20) it follows that

$$-2\tilde{U}_N + \lambda_{BN+1}\tilde{U}_{N+1} = \tilde{W}_{N+1} + \tilde{\varphi}_1 = \tilde{W}'_{N+1} \tag{3.21}$$

$$-2\tilde{U}_1 + \lambda_{B0}\tilde{U}_0 = \tilde{W}_0 + \tilde{\varphi}_0 = \tilde{W}'_0 \tag{3.22}$$

We are seeking a solution of the form:

$$u_{i+1} = A_i U_i + R_i \text{ or } \tilde{U}_{i+1} = \lambda_{Ai}\tilde{U}_i + \tilde{R}_i \tag{3.23}$$

Considering (3.23) for i = N and from (3,21) it follows that

$$A_{AN} = 2\lambda_B^{-1} \text{ and } \tilde{R}_N = \lambda_B^{-1}\tilde{W}'_{N+1} \tag{3.24}$$

and considering (3.23) for i = 0 and from (3,22) we get

$$\tilde{U}_0 = (\lambda_B - 2\lambda_{A0})^{-1}(2\tilde{R}_0 + \tilde{W}'_0) \tag{3.25}$$

Starting with A_{AN} and $\tilde{R}_N$ given by (3.24), λ_{Ai} and $\tilde{R}_i$, i = N-1 to 0, can be computed as before from (3.7)-(3.8). Once λ_{A0} and $\tilde{R}_0$ are computed, $\tilde{U}_0$ can be obtained from (3.25) and then $\tilde{U}_i$, for i = 1 to N, can be computed from (3.23). As can be seen, except for the use of matrices $\Theta\mathcal{Y}$ and $\mathcal{Y}^{-1}\Theta$ for performing the direct and inverse DCT, the above procedure differs from that of Dirichlet boundary condition of §3.2 only in computation of A_{AN}, $\tilde{R}_N$, and $\tilde{U}_0$.

Note that, since $\lambda_{Bj} > 2$ for j = 0 to N, from (3.24) we have $\lambda_{AN,j} < 1$. It can be then easily shown that $\lambda_{Ai,j} < 1$ for all i and j = 0 to N. Thus, we have

$$\lambda_{Bj} - 2\lambda_{A0j} > 0$$

which proves that the diagonal matrix in (3.25) can be inverted.

4. Parallel Implementation of Fast Invariant Imbedding Algorithms

In this section we discuss the performance of the Fast Invariant Imbedding algorithm for parallel computation. We compare this performance with that of MD algorithm while implemented on the same architecture.

4.1. Fine Grain Parallel Computation: Time and Processor's Bounds

With $O(N^2)$ processors, the computation of the MD algorithm can be performed in $O(\text{Log } N)$ [12] as follows. In Steps 1 and 5 the N DSTS can be computed in parallel. Each DST can be performed in $O(\text{Log } N)$ with O(N) processors [12]. Thus, by using $O(N^2)$ processors, the cost of parallel computation of Steps 1 and 5 is of $O(\text{Log } N)$. In Step 3, N tridiagonal systems can be solved in parallel. With O(N) processors, each tridiagonal system can be solved in $O(\text{Log } N)$, by using, for example, the parallel algorithms in [131. Thus, with $O(N^2)$ processors, the cost of parallel computation of Step 3 is $O(\text{Log } N)$.

The same time- and processor-bounds can be also achieved for the Fast Invariant Imbedding algorithm. Step 1 involves the evaluation of N decoupled CFS (§3.3). With O(N) processors and by using the algorithm in [111, each CF can be computed in $O(\text{Log } N)$. Thus, by using $O(N^2)$ processors, the cost of parallel computation of this step is of $O(\text{Log } N)$. In Step 2, the two vector additions for computation of W_1' and W_N' can be performed in O(1) by using O(N) processors. The rest of the computation in Step 2 and the computation of Step 5 are exactly the same as in Steps 1 and 5 of the MD algorithms and thus can be performed in $O(\text{Log } N)$ by using $O(N^2)$ processors. The vector recurrences in Steps 3 and 4, as shown in §3.3., can be decomposed into a set of N decoupled scalar FOILRs. With O(N) processors and by using the algorithms in [11,13], each FOILR can be computed in $O(\text{Log } N)$. Thus, with $O(N^2)$ processors, the complexity of parallel computation of Steps 3 and 4 is of $O(\text{Log } N)$.

However, achieving the time lower bound of $O(\text{Log } N)$ in parallel computation of either the MD algorithm or the Fast Invariant Imbedding algorithm requires an excessive number of processors. More important, in order to limit the communication complexity to $O(\text{Log } N)$, a very complex processors interconnection is required. In the following, we consider a more realistic coarse grain parallel implementation of both algorithms.

4.2. Coarse Grain Parallel Computation

We consider a coarse grain parallel computational strategy by using N processors. It should be mentioned that the early interest in the MD algorithm was motivated by its theoretical efficiency for parallel implementation with N processors [4. In fact, with N processors, the computation of N decoupled DSTS in Steps 1 and 5 can be performed fully in parallel with a complexity of $O(N\text{Log } N)$. That is, the cost of parallel implementation of Steps 1 and 5 is equal to **that of one DST. Similarly, the N decoupled linear systems in Step 3 can be performed in parallel. Thus,** the cost of parallel computation of Step 3 is equal to that of one single linear system solution. *Thisimplies* a perfect ***linear speedup of N in the computation by*** using N ***processors.***

However, a close examination shows that the resulting communication cost can significantly degrade the overall performance of such parallel computation strategy. To see this, recall that the operation in Steps 2 and 4 corresponds to transposing matrices $\tilde{\underline{W}}$ and $\hat{\underline{U}}$. The communication complexity of matrix transposition is a function of the processors interconnection structure. With

N processors interconnected with a perfect shuffle or a Hypercube topology, the communication complexity of matrix transposition is of $O(N\text{Log } N)$ [14,15]. This implies that, asymptotically, the computation and communication costs are the same. Obviously, on architectures with simpler interconnection topologies, e.g., linear array or mesh, the communication cost will be much greater than the computation cost.

However, the practical implementation on MIMD architectures with even Hypercube topology can result in an actual communication cost much greater than the computation cost. To see this, let β and α denote the cost of the communication start-up (or, latency) and the elemental transfer time. Note that, usually, α is approximately equal to the cost of one floating-point operation, i.e., f, while $\beta \gg \alpha$ [16]. The communication cost of Steps 2 and 4, by using the algorithm in [15], is then given by $2(\beta+\alpha)(N\text{Log } N)$. Neglecting the lower order terms, the computation cost of Steps 1 and 5 is given by $5f(N\text{Log } N)$. Since β is much greater than f (even by as much as two orders of magnitude for many commercially available Hypercube architectures), it follows that the communication cost of the algorithm can be indeed much greater the computation cost.

Now, let us consider the parallel implementation of the Fast Invariant Imbedding algorithm by N processors, denoted by PR_i for i = 1 to N. In Step 2, the computation of W'_1 by PR_1 and W'_N by PR_N can be performed in parallel with a cost of O(N). The rest of the computation of Step 2 as well as that of Step 5 can be performed similar to that of Steps 1 and 5 of the MD algorithm with a cost of $O(N\text{Log } N)$ in fully parallel fashion and without any communication among processors. By using the parallel algorithms in [11], the computation of CFS in Step 1 and the FOILRs in Steps 3 and 4 can be computed in $O(N\text{Log } N)$. On an architecture with a perfect shuffle interconnection, the communication complexity of such a strategy for parallel computation of Steps 1, 3, and 4 is of $O(N\text{Log } N)$. Thus, on an MIMD architecture with the perfect shuffle topology, the communication complexity of the Fast Invariant Imbedding algorithm will be of $O((\beta+\alpha N)\text{Log } N)$. For typical values of N of the order of hundred, this represents about two orders of magnitude improvement in the communication cost over that of the MD algorithm on the same architecture.

The simple communication structure of the Fast Invariant Imbedding algorithm enables its efficient implementation on architectures with even simpler processors interconnection topology, To see this, let us consider the implementation of the algorithm on an MIMD architecture with a simple nearest neighbor interconnection (Fig. 1). As is shown in [13], with a nearest neighbor interconnection the communication complexity of parallel computation of Steps 1, 3, and 4 will be of $O((\beta+\alpha N)N)$. Insofar as $\beta \gg \alpha$ and for the values of N in the range of hundreds, this represents a major improvement over the communication cost of the MD algorithm implemented with a hypercube topology.

The Intel *i860* and DEC Alpha are representatives of an emerging class of single chip processors with vector processing capability. Such vector processors are increasingly used as processing nodes in massively parallel MIMD architectures, e.g., Intel Touchstone Delta and Paragon, and CRAY T3D. If the architecture of Fig. 1 is implemented by using such low-cost and powerful vector processors then a further speedup in computation of the Fast Invariant

Imbedding algorithm can be achieved. This follows from the fact that parallel computation of Steps 1, 3, and 4 involves operations on long vectors and hence is highly efficient for vector processing,

4.3. Algorithmically Specialized Parallel Architectures for Implementation of Fast Invariant Imbedding Algorithm

4.3.1 A Communication Efficient Variant of Fast Invariant Imbedding Algorithm

It is possible to further reduce the communication cost of the Fast Invariant Imbedding algorithm by using a hybrid parallel/pipelined computational strategy. To see this, note that, the above discussed N-parallel strategy is based on parallel computation of Step 1, with diagonal matrices λ_{Al} as the basic data, and Steps 3 and 4, with vectors $\tilde{R}_i$ and $\tilde{U}_i$ as the basic data. However, as shown in Eq. (3.12), the computation of Step 1 can be reduced to that of a set of CFS. Similarly, as shown by Eqs. (3.10)-(3. 11), the computation of Steps 3 and 4 can be reduced to that of a set of scalar FOILRs. An efficient hybrid parallel/pipelined computational strategy is then based on parallel computation of Steps 2 and 5 as before but pipelining in the computation of the set of CFS in Step 1 and the sets of scalar FOILRs in Steps 3 and 4. In order to describe this pipelining strategy, let us again consider an implementation by using N processors, denoted by PR_i for i = 1 to N. The activities of processors PR_i for computing Steps 1, 3, and 4 is then given as

Step 1

For j = 1 to N, Do

  Receive $\lambda_{Ai,j}$ from PR_{i+1}

  Compute $\lambda_{Ai-1,J} = I/(a^*, - \lambda_{Ai,'})$

  Send $\lambda_{Ai-1,j}$ to PR_{1-1}

End_Do

Step 3

For j = 1 to N

  Receive $\tilde{R}_{i,j}$ from PR_{i+1}

  Compute $\tilde{R}_{i-1,j} = \lambda_{Ai-1,j}(\tilde{R}_{i,j} + \tilde{W}'_{i,j})$

  Send $\tilde{R}_{Al-l,j}$ to $PR_{i\,1}$

End_Do

Step 4

For j = 1 to N, Do

  Receive $\tilde{U}_{i-1,j}$ from $PR_{i\,1}$

  Compute $\tilde{U}_{i,j} = \lambda_{Ai-1,j}\tilde{U}_{i-1,J} + \tilde{R}_{i-1,j}$

  Send $\lambda_{Ai-1,j}$ to $PR_{i\,1}$

End_Do

With this pipelining strategy, the complexity of computation of Steps 1, 3, and 4 is of O(N) which indicates a speedup of O(N). More important, is the fact that, by overlapping the computation and communication, the communication cost is now of O(1) while only demanding data transfer between Adjacent processors. Note, however, that this parallel/pipeline strategy leads to a

heterogeneous computational structure since it involves highly coarse grain computations in Steps 2 and 5 and very fine grain computations in Steps 1, 3, and 4. In fact, the key to an efficient implementation of this pipeline strategy is the capability to perform fine grain computation as well as fast nearest neighbor communication. In this regard, this strategy is not suitable for implementation on the MIMI) architecture of Fig. 1 since it can not perform nearest neighbor communication in a fast and efficient way. In the following, we discuss two more optimal architectures for implementation of this parallel/ pipeline strategy,

4.3.2. An Optimal Algorithmically Specialized Parallel Architecture

The computation of various Discrete Transforms (DTs) arises in many engineering and scientific applications. This has motivated the development of special-purpose chips for fast computation of DTs by both academia and industry. These chips achieve an optimal performance in performing DTs by customizing the hardware architecture for the specific application and by exploiting a high degree of parallelism in the computation. An extensive survey of such special-purpose chips for performing DCT (which also arises in data compression applications) is presented in [17]. We have also presented a new hardware technology based on charge domain computing devices [18] which is particularly efficient for performing various DTs since it is capable of exploiting a massive degree of parallelism in the computation.

A heterogeneous, algorithmically-special ized, parallel architecture for optimal implementation of the Fast Invariant Imbedding algorithm can be then designed by using special-purpose chips for performing the DCTs in Steps 2 and 5, and fine-grain processors, such as Digital Signal Processor (DSP) chips, for computation of Steps"l, 3, and 4. Figure 2 shows such an architecture. The optimality of this architecture for the Fast Invariant Imbedding algorithm follows from the fact that, in addition to efficient implementation of the parallel/pipeline strategy and thus minimizing the communication cost, it allows a fast computation of Steps 2 and **5 by using special-purpose** chips and exploiting further parallelism in computation of DCTS.

4.3.3. A Low Cost Algorithmically Specialized Parallel Architecture

The development of DSP chips was mainly motivated for computing DTs. The DSP chips only employ a pipelined architecture to speedup the computation of DTs. In this regard, they can not achieve the optimal performance of other special-purpose chips which exploit a high degree of parallelism in the computation. Nevertheless, the DSP chips can be used both as fine-grain processors for performing Steps 1, 3, and 4 and as coarse-grain processors for performing the DCTS (though with less optimal performance).

This suggests that a linear array of DSP chips (Fig. 3) can also be used for efficient implementation of the parallel/pipeline strategy for the Fast Invariant Imbedding algorithm, Although, compared with the architecture of Fig. 2, this architecture achieves a less optimal performance in computing Steps 2 and 5, it represents a much more cost-effective alternative with a greater ease for the design and implementation.

ACKNOWLEDGMENT

This ***research*** **was** ***carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration (NASA), The authors gratefully acknowledge the support and encouragement of Dr. George Sevaston from JPL.***

REFERENCES

1. M. Milman, A. Fijany, and D. Redding, "Wavefront Control Algorithms for a Dense Adaptive Optics System, " Proc. SPIE Int. Symp. OE/LASE 94 (this proceeding), Los Angeles, CA, Jan. 1994.
2. B. Buzbee, G. Golub, and C. Nielson, "On Direct Methods for Solving Poisson Equations, " SIAM J. Numer. Anal., Vol. 7, pp. 627-656, 1970.
3. P. Swarztrauber and R. Sweet, "Vector and Parallel methods for the Direct Solution of Poisson's Equation, " J. Computional & Applied Math., Vol. 27, pp. 241-263, 1989.
4. B. Buzbee, "A Fast Poisson Solver Amenable to Parallel Computation, " IEEE Trans. Computers, Vol. C-22, pp. 793-796, 1973.
5. R. Sweet, W. Briggs, S. Olivera, J. Porsche, and T. Turnbull, "FFTs and Three-Dimensional Poisson Solvers for Hypercube, " Parallel Computing, vol. 17, pp. 121-131, 1991.
6. E. Angel, "A Building Block Technique for Elliptic Boundary-Value Problems over Irregular Regions, " J. Math. Anal. Appl. , Vol. 26, pp. 75-81, 1969.
7. E. Angel and R. Bellman, ***Dynamic Programming and Partial Differential Equations.*** Academic Press, 1972.
8. S. Barnett, ***Matrices: Methods and Applications.*** Clarendon Press, 1990.
9. C. Van Loan, ***Computational frameworks for the Fast Fourier Transform.*** SIAM, Philadelphia 1992.
10. R.F. Boisvert, ''Algorithms for Special Tridiagonal Systems, " SIAM J. Sci. Stat. Comput. , Vol. 12(2), pp. 423-442, March 1991.
11. P.M. Kogge and H.S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations, " IEEE Trans. on Computers, Vol. C-22(8), pp. 786-793, Aug. 1973.
12. A. H, Sameh, S. C. Chen, and D. J. Kuck, "Parallel Poisson and Biharmonic Solvers, " Computing, Vol. 17, pp. 219-230, 1976.
13. R. Hockney and C. Jesshope, ***Parallel Computers.*** Adam Hilger Ltd. , 1981.
14. H.S. Stone, ''Parallel Processing with the Perfect Shuffle, " IEEE Trans. on Computers, Vol. C-20(2), pp. 153-161, Feb. 1971.
15. 0. McBryan and E. Van De Velde, "Hypercube Algorithms and Implementation, " SIAM J. Sci. Stat. Comput. , Vol. 8(2), pp. 227-287, March 1987.
16. Y. Saad and M.H. Schultz, ''Data Communication in Hypercube, " J. Parallel and Distributed Computing, Vol. 6, pp. 115-135, 1989.
17. K.R. Rao **and P. Yip,** ***Discrete Cosine Transform: Algorithms, Advantages,*** Applications. Academic Press, 1990.
18. A. Fijany, J. Barhen, and N. Toomarian, ''Massively Parallel Neurocomputing for Aerospace Applications, " Proc. Computing in Aerospace 9 Conf. , pp. 1002-1010, San Diego, CA, Oct. 1993,

Figure 1. An MIMD Parallel Architecture with Nearest Neighbor Interconnection

VP: Vector Processor, e.g., Intel i860, DEC Alpha

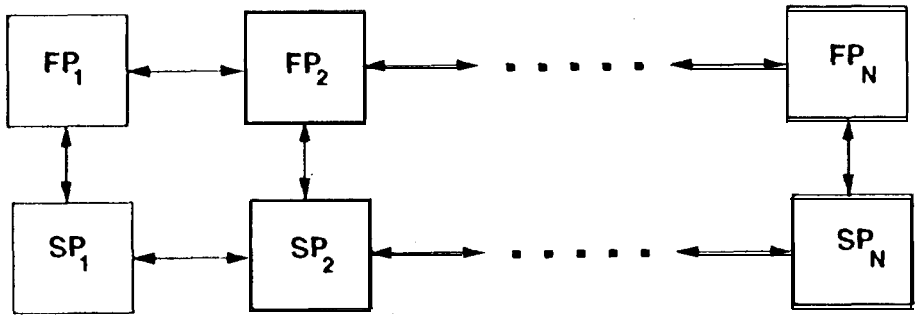

Figure 2. An Optimal Algorithmically Specialized Parallel Architecture

FP: Fine Grain Processor, e.g., a DSP Chip SP: Special-Purpose Processor for performing DCT

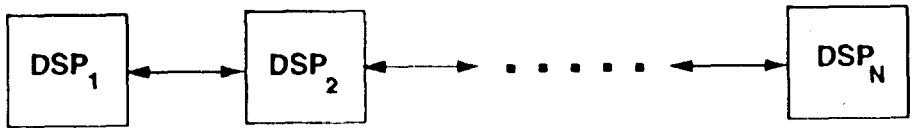

Figure 3. A Linear Array of Low-Cost DSP Chips

DSP: Digital Signal Processor Chip